

# ILP Modulo Theories

Panagiotis Manolios

Northeastern University

pete@ccs.neu.edu

Vasilis Papavasileiou

Northeastern University

vpap@ccs.neu.edu

## Abstract

We present Integer Linear Programming (ILP) Modulo Theories (IMT). An IMT instance is an Integer Linear Programming instance, where some symbols have interpretations in background theories. We develop the theoretical underpinnings for IMT by means of  $BC(T)$ , the Branch and Cut Modulo  $T$  abstract transition system. We show that  $BC(T)$  provides a sound and complete optimization procedure for the ILP Modulo  $T$  problem, as long as  $T$  is a decidable, stably-infinite theory.

Our motivation for developing IMT is to enable advances in synthesis by coupling high-level declarative languages with new, powerful constraint solving and optimization techniques. This allows designers to specify *what* they want, not *how* to achieve it. We have applied the IMT approach to industrial synthesis and design problems with hard real-time constraints arising in the development of the Boeing 787. Given that many practical problems ranging from operations research to software design can be thought of as synthesis problems, we believe that our framework has the potential to be widely applicable.

**General Terms** Design, Languages

**Keywords** Declarative Programming, Constraint Programming, Decision Procedures, Satisfiability Modulo Theories, Linear Programming, Optimization, Extensible Languages

## 1. Introduction

Managing the complexity of modern software-intensive systems requires raising the level of discourse by utilizing abstraction, in particular high-level modeling. For example, Boeing’s 787 Dreamliner consists of thousands of interacting components, and just the software enabling fly-by-wire control is over 10 million lines of code. To design such complex systems, numerous models at various levels of abstraction are used. The highest level models are *architectural* models; they describe structural properties and component interactions. While many architecture description languages have been proposed, these languages require users to specify what components are to be used and how they are to be connected. The effort required to do this can be significant. According to Boeing engineers, during the design of the 787 Dreamliner, the construction of the architectural models required the “cooperation of multiple teams of engineers working over long periods of time.”

Recently, we developed the CoBaSA (Component-Based System Assembly) framework. Using CoBaSA, we algorithmically synthesized architectural models using the actual production design data and constraints arising during the development of the Dreamliner [19, 29]. CoBaSA provides a high-level modeling and specification language, coupled with a synthesis tool based on verification and optimization technology. Key to the success of our approach was the combination of ILP with a custom decision procedure for real-time scheduling. We were able to automatically synthesize architectures in minutes directly from the high-level requirements. This allows designers to specify *what* they want, not *how* to achieve it. It enables deep design-space analysis and exploration at the earliest stages of design, when this kind of analysis is most useful. Finally, our work makes it possible to analyze and respond to disruptive events, such as changes to requirements, in real time.

In many ways, Boeing’s 787 Dreamliner is one of the most complex systems ever built, and our success in fully automating significant, industrially-relevant design problems is compelling evidence of the potential advances in synthesis that new constraint solving techniques can enable. To be more broadly successful, new approaches to synthesis should be both *expressive* enough to model design problems arising in a variety of real systems and *scalable* enough to automatically synthesize solutions. It is difficult to make the case with a theoretical argument, since synthesis is *hard* (NP-Hard) even with very simple classes of constraints, and very quickly becomes undecidable [25]. In this paper, we show that while our framework has been primarily used for component-based design, it can also be applied to different problems in fields varying from operations research to database design. In fact, CoBaSA has been designed with extensibility in mind and allows for domain-specific modeling constructs. CoBaSA is publicly available.<sup>1</sup>

To be widely applicable, we desire a *parametric* framework that can be instantiated for the particular class of synthesis problems. To this end, we propose the Integer Linear Programming (ILP) Modulo Theories (IMT) framework for combining ILP with background theory solvers. We can use Mixed-Integer Programming (MIP) instead of ILP, but in this paper we will focus on ILP for simplicity. By relying on ILP as the core, we build upon more than five decades of intensive research in integer programming [17]. In addition, the ILP emphasis on optimization (as opposed to feasibility) is of fundamental importance in synthesis, as solution spaces are typically very large, and the goal is to find solutions that are optimal with respect to certain design objectives.

The primary goal of this paper is to introduce the theoretical underpinnings of IMT. We do that via the  $BC(T)$  framework (Branch and Cut Modulo  $T$ ).  $BC(T)$  models the branch-and-cut family of algorithms for integer programming as an abstract transition system, and allows plugging in theory solvers. Building on classical results on combining decision procedures [26, 41], we show that

[Copyright notice will appear here once ‘preprint’ option is removed.]

<sup>1</sup> <http://www.ccs.neu.edu/home/vpap/cobasa.html>

$BC(T)$  provides a sound and complete optimization procedure for the combination of ILP with stably-infinite theories.

$BC(T)$  is the IMT counterpart to the DPLL( $T$ ) architecture for lazy SMT [36]. We study the relationship between the two systems and show that  $BC(T)$  can simulate DPLL( $T$ ), *i.e.*, a  $BC(T)$ -based solver can be used as a drop-in replacement for an SMT solver that supports Quantifier-Free Linear Integer Arithmetic (QF\_LIA).

**Contributions** We provide the first to our knowledge framework for combining ILP with theories (IMT). We provide a general  $BC(T)$  architecture for solving IMT problems. We prove that  $BC(T)$  provides a sound and complete optimization procedure for the combination of ILP with a stably-infinite theory  $T$ .

**Paper Structure** The rest of the paper is organized as follows. Section 2 introduces the CoBaSA synthesis paradigm by means of examples. In Section 3, we formally define IMT and provide an abstract  $BC(T)$  architecture for solving IMT problems. Section 4 elaborates on the relationship between SMT and IMT. Section 5 contains experimental evidence that the combination of ILP and theories is crucial for the practicality of CoBaSA. We provide an overview of related work in Section 6, and conclude with Section 7.

## 2. Synthesis Paradigm

CoBaSA was initially developed to tackle problems that arise in the development of large-scale, component-based systems. The engineers building a large-scale system, *e.g.*, a commercial airplane, have access to a pool of components. Their job is to select some of these components, connect, integrate, and assemble them in a meaningful way. We call this the *system assembly* problem. The CoBaSA modeling language provides object-oriented constructs to describe the components of a system, and also declarative ways to impose constraints on how these components will inter-operate. The synthesis backend is responsible for finding a way to select and integrate the components in a way that meets the high-level specification.

Our modeling language and constraint solving engine are general enough to be useful in other classes of applications, not necessarily involving component-based systems. In this section, we will provide an introduction to the CoBaSA language by means of examples arising in different domains. Our examples demonstrate how the system can be augmented for domain-specific needs. We introduce CoBaSA concepts on demand.

### 2.1 Applications

#### 2.1.1 TA Allocation

We are given a pool of teaching assistants (TAs), a set of instructors, and a set of courses. Instructors have positive preferences, *e.g.*, they frequently want to work with their advisees. They also have negative preferences, as they may have negative impressions of some TAs. Each course requires a fixed number of TAs. We have to determine which course each TA will participate in.

The CoBaSA language is typed. The basic datatypes provided are integers (optionally bounded), Booleans, and strings. *Entities* resemble classes in an object-oriented language. The entity definitions of Figure 1 reflect the problem setup. Instructor and TA entities have a single “id” field (which can be their name). Each course has an integer as the identifier (course number), a name, a list of instructors, and a bounded integer for the required number of TAs (field *needs*). A course also carries a list of instructor preferences, for which we need a helper entity. The idea is that instructors assign points to denote preferences, with the most desirable TA getting the maximum number of points.

With entity definitions in place, we proceed to describe the actual data (Figure 2). We model two of the instructors in our

```
entity instructor {
  id: string
};

entity ta {
  id: string
};

entity preference {
  ta: ta;
  points: bounded 1 to 6
};

entity course {
  id: int;
  name: string;
  instructors: list instructor;
  needs: bounded 0 to 10;
  preferences: list preference
};
```

Figure 1: TA Allocation Entities

```
var manolios      = instructor{"Pete Manolios"};
var wahl          = instructor{"Thomas Wahl"};
var ramachandran = ta{"Jaideep Ramachandran"};

var cs2800 =
  course{
    2800; "Logic and Computation";
    [manolios; wahl]; 4;
    [preference{ramachandran; 6}; ...]
  };

var tas          = [ramachandran; ...];
var courses      = [cs2800; ...];
```

Figure 2: TA Allocation Data

department, Manolios and Wahl. We also instantiate the *ta* entity for Ramachandran. We are then ready to describe the “Logic and Computation” course (CS2800), taught by Manolios and Wahl. CS2800 requires four TAs. The instructors do their best to get Ramachandran in the team, *i.e.*, he gets 6 preference points. We group TAs and courses in lists.

```
map m tas courses [] [];

forall c in courses:
  c.needs = sum ta in tas: m(ta, c);

min -(sum c in courses:
  sum p in c.preferences:
    m(p.ta, c) * p.points);
```

Figure 3: Defining a map from TAs to Courses

Our goal is to assign exactly one course to each TA. We achieve this with the CoBaSA *map* statement (Figure 3). *map* existentially defines a function whose domain is the list of TAs and whose range is the list of courses. We subsequently constrain the mapping so that each class will get the required number of TAs. The sub-language for expressing constraints supports arbitrary Boolean combinations

of linear arithmetic constraints. It also provides facilities for ranging over all elements of a list, either for succinctly applying a constraint to all elements (`forall`), or for adding up an expression (`sum`). The *map reference*  $m(\text{ta}, c)$  is true iff  $\text{ta}$  is mapped to  $c$ . It is valid for a Boolean expression like  $m(\text{ta}, c)$  to appear within an arithmetic expression, behaving as a  $\{0, 1\}$ -integer.

CoBaSA is built on top of optimization technology, and thus allows programmers to provide objective functions. For TA allocation, our objective is to keep instructors as happy as possible. The objective function of Figure 3 maximizes the sum of preference points for the TA assignments that actually match the instructors' wishes (we maximize a quantity by minimizing its negation). We sum over all courses, and over all elements of the preference list associated with each course.

The problem is easy to tackle with CoBaSA: the language provides the right abstractions and the ILP solver has no trouble finding an optimal solution quickly. We have used CoBaSA successfully to find a TA allocation for the Spring 2012 semester in our department. The requirements took less than a person-hour to encode. CPLEX [2] needed less than a second to find an optimal solution.

### 2.1.2 Database Debugging

			entity emp_schema {
			id: int;
			age: bounded 0 to 120;
			salary: bounded 0 to 20000
			};
<b>id</b>	<b>age</b>	<b>salary</b>	var x: bounded 0 to 120;
0	32	3000	var y: bounded 0 to 20000;
1	28	2800	
2	30	3200	
3	41	4500	var employees =
4	x	y	[emp_schema{0; 32; 3000};
			emp_schema{1; 28; 2800};
			emp_schema{2; 30; 3200};
			emp_schema{3; 41; 4500};
			emp_schema{4; x; y}];

Figure 4: Database Table Representation

Database users frequently encounter the following scenario: they perform a query, but the output table does not contain some of the expected results. Sometimes users even get no output tuples at all.

Constraint solving is a good fit for debugging problematic queries. If the output of a query falsifies a given specification, we can try to *synthesize* one or more additional input tuples that translate to output tuples with the desired characteristics. By providing such tuples we prove that the query is not altogether inconsistent.

We show how to embed database concepts in CoBaSA. As a first step, we want to be able to describe database tables. A *table schema* (i.e., the field names and types of a table) can be sufficiently described with a CoBaSA entity definition. Tuples in a certain table are encoded as instances of the appropriate table schema; tables are just lists of instances. Figure 4 demonstrates how we can represent a table of employees. Note that the table contains symbolic data, in particular an employee with unknown age and salary.

Describing database queries is harder. Instead of trying to express queries in terms of the core language constructs, we utilize the framework's syntax extension mechanism to define a Domain-Specific Language (DSL) that provides database operators like selection, projection, union, join, and cross-product. The database query of Figure 5a now becomes valid CoBaSA syntax (we produce a table named `output` with all employees aged at most 35 whose salary is at least \$3500). Note that the operators we define

```
var output =
  % select x from employees
  where x.age <= 35 and x.salary >= 3500;

(a) Query

% exists x in output: true;

(b) Table Constraint

var limit: bounded 0 to 120;

var output =
  % select x from employees
  where x.age <= limit and x.salary >= 3500;

min limit;

(c) Query with Symbolic Values
```

Figure 5: Domain-specific Syntax for Database Queries

do not just perform evaluation with concrete values (like a DBMS would), but work with a mix of concrete and symbolic values. We then impose a constraint on the output table (Figure 5b): in our simple example, the output table shouldn't be empty. CoBaSA looks for appropriate input values for the record with `id` 4 so that the requirements will be met: it comes up with an employee who is 35 years old and earns \$3500 a month.

Instead of coming up with new tuples, the user may want us to fix the query itself, i.e., synthesize a query that is reasonably close to the original, but produces the expected answer. For example, we can relax some conditions so that more tuples will go through. In our example, we could let CoBaSA find a different age limit. In Figure 5c we achieve this by providing a query skeleton, which will turn into a concrete query once we find a value for `limit`. Optimization capabilities are crucial, because showing that 120 is an appropriate age limit would be trivial but not useful.

```
val plugin_select:
  id -> texpr -> texpr -> env -> texpr

let extend e =
  EXTEND
  e: FIRST
  [ [ "%"; "select"; i = LIDENT;
    "from"; e1 = SELF;
    "where"; e2 = SELF ->
    E_Ext (plugin_select i e1 e2) ]
    ...
  ];
END
```

Figure 6: Anatomy of the Syntax Extension

To allow embedding DSLs, we build upon Camlp5 Extensible Grammars [1]. Figure 6 gives a glimpse of how we actually extend the CoBaSA grammar for databases. The non-terminal symbol for expressions (`e`) is extended with a new production for database selection. We define the semantics of this extension by means of a compiler "plugin", the function `plugin_select`. In the fragment of code we show, `plugin_select` is partially applied: its last argument is the environment. When this closure is finally applied on the appropriate environment argument, it will produce a term

in the internal representation of expressions (type `texpr`), *i.e.*, `plugin_select` desugars the `select` operator.

### 2.1.3 Architectural Synthesis with Real-Time Constraints

We applied CoBaSA to solve the architectural synthesis problem for the real, production data from the 787 Dreamliner, which was provided to us by Boeing [19]. The basic components for this problem are anywhere from 8 to 22 cabinets (providing resources like CPU time, bandwidth, battery backup, and memory), about 200 software applications that reside on the cabinets (and consume resources), and up to 300 global memory spaces (that also consume resources). Applications communicate via a publish-subscribe network where messages are aggregated into virtual links and transmitted via multicast.

The core CoBaSA language suffices to model all classes of resource and architectural requirements. For illustration purposes, we provide a scaled-down example (Figure 7) that retains the flavor of the actual models. We focus on applications and cabinets, and omit other kinds of components; we also ignore resources other than RAM and CPU time. We start by defining the entities (Figure 7a). We then proceed to instantiate the entities. We have four concrete cabinets, each of which provides 256MB of RAM and 1000000 CPU cycles per second. We also have 8 applications. We subsequently organize applications and cabinets in lists (Figure 7c).

Each application has to be mapped to (*i.e.*, reside on) exactly one cabinet (Figure 7d). The mapping is subject to resource requirements: the applications that co-reside on a cabinet cannot exceed the provided amount of CPU and RAM. The map statement provides a shorthand for expressing such resource constraints. The map `m` is further constrained to satisfy other (*i.e.*, non-resource) requirements. For example, applications `t7_0` and `t7_1` are two copies of the same safety-critical process; they have to reside on different cabinets to achieve fault-tolerance. This is a separation constraint, and can be expressed as shown in Figure 7e. Similarly, we can have co-location constraints, components fixed on specific cabinets, etc.

Interesting architectural synthesis instances from the aerospace domain also have optimization criteria. For example, we may want to minimize the maximum bandwidth consumption across cabinets, *i.e.*, balance the load. The rationale is that the requirements may change in the future (*e.g.*, a component supplier is unable to meet the agreed-on resource consumption), but it is not always possible to change the design, for instance because the cost of recertification is astronomical. When this happens, some headroom in resource availability is precious.

The architectural models we synthesize also have to satisfy hard real-time constraints. The scheduling discipline (static cyclic) is non-preemptive and very hard to satisfy in practice. This class of constraints is not easy to model with the core CoBaSA language. In fact, the problem is deeper, as we do not know how to reduce static cyclic scheduling constraints to ILP constraints in a way that leads to reasonable solving times by modern ILP solvers. Therefore, we cannot follow the approach of Section 2.1.2, *i.e.*, compile higher modeling constructs to the core constructs, without touching the underlying formalism.

Instead what we need is an extension to the underlying logic (integer linear constraints) and the corresponding solver. We have implemented a custom decision procedure for scheduling, and a mechanism for this decision procedure to communicate with the ILP solver [19]. The backend now interleaves ILP solving and scheduling, *i.e.*, we have an instance of the ILP Modulo Theories framework. In each iteration, CoBaSA obtains a satisfying assignment for an ILP instance. If the assignment is consistent with the scheduling requirements, CoBaSA reports an answer. Otherwise,

CoBaSA learns a theory lemma that precludes the unschedulable allocation, along with a class of similar unschedulable allocations.

The backend extension gives rise to a schedulability predicate that appears in the models: `sched(m, tasks, cabinets)` (where `m` is a map and `tasks, cabinets` are lists) is true iff the tasks co-located on each cabinet according to the map `m` are schedulable. In other words, extending the underlying formalism is reflected on the modeling language.

## 2.2 Discussion

We described different applications of the CoBaSA language. All three applications give rise to integer linear constraints. In all cases optimization is required, or at least strongly desired. Integer arithmetic and optimization requirements point to ILP as a suitable backend for synthesis. We experimentally show that ILP indeed works better than the alternatives in Section 5.

The applications differ with respect to their language requirements. (a) For TA allocation, core CoBaSA suffices. (b) Database debugging requires domain-specific syntax. (c) For architectural synthesis, we extend the backend with a custom decision procedure. It is possible for an application to require extensions in both directions. For example, we can have an aerospace DSL with separation and co-location operators (among others), combined with a background theory for scheduling. The separation constraint shown in Figure 7e can then be written more succinctly as `sep(m, t7_0, t7_1)`.

From a decision procedure perspective, extending the language without touching the ILP backend resembles the *eager* approach to SMT [16, 28, 40], *i.e.*, compiling high-level constructs like bitvector operations to SAT. Coupling an ILP solver with an external decision procedure resembles the *lazy* SMT architecture [5, 11, 36]. Which of the two approaches is better depends on the problem structure and size. The lazy approach is harder to implement, but also tends to scale better. For example, eager translation suffices for our small database examples; larger-scale problems may require efficient specialized data structures (*e.g.*, indices), and thus mandate a background solver.

For the rest of the paper, we focus on the lazy approach. A general framework for combining ILP with theories promises advances in synthesis. We will therefore define an ILP Modulo Theories (IMT) framework, and provide an abstract  $BC(T)$  architecture for lazy IMT solving.

## 3. $BC(T)$

In this section, we formally define IMT. We also provide a general  $BC(T)$  architecture for solving IMT problems. We describe  $BC(T)$  by means of a *transition system*, similar in spirit to the  $DPLL(T)$  transition system [36]. It is intended to describe how to combine an ILP solver and a background decision procedure for a theory  $T$  to obtain a solver for ILP Modulo  $T$ .  $BC(T)$  models and generalizes (a) the branch-and-cut family of algorithms, (b) the lazy approach to Satisfiability Modulo Theories, and (c) the non-deterministic Nelson-Oppen combination of decision procedures, where one of the theories is Linear Integer Arithmetic.

### 3.1 Formal Preliminaries

An *integer linear expression* is a sum of the form  $c_1 v_1 + \dots + c_n v_n$  for integer constants  $c_i$  and variable symbols  $v_i$ . An *integer linear constraint* is a constraint of the form  $e \bowtie r$ , where  $e$  is an integer linear expression,  $r$  is an integer constant, and  $\bowtie$  is one of the relations  $<$ ,  $\leq$ ,  $=$ ,  $>$ , and  $\geq$ . An *integer linear formula* is a set of (implicitly conjoined) integer linear constraints. We will use propositional connectives over integer linear constraints and formulas as appropriate, and omit  $\wedge$  when this does not cause

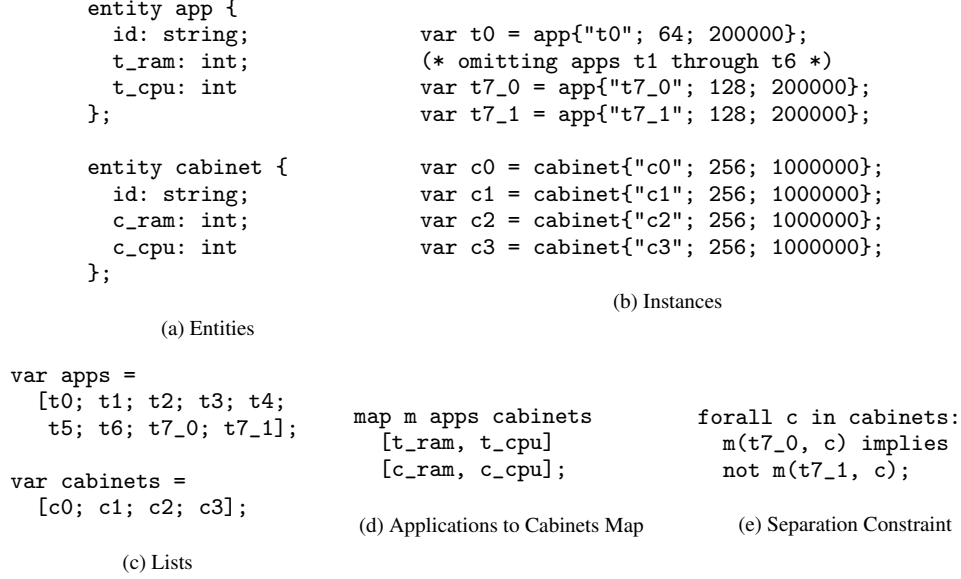


Figure 7: Avionics Architectural Synthesis Model

ambiguity (*i.e.*, juxtaposition will denote conjunction). An *integer linear programming (ILP) instance* is a pair  $C, O$ , where  $C$  is an integer linear formula and the *objective function*  $O$  is an integer linear expression. Our goal will always be *minimizing* the objective function.

We assume a fixed set of variables  $\mathcal{V}$ . An integer assignment  $A$  is a function  $\mathcal{V} \rightarrow \mathbb{Z}$ , where  $\mathbb{Z}$  is the set of integers. We say that an assignment  $A$  satisfies the constraint  $c = (c_1 v_1 \dots c_n v_n \bowtie r)$  (where  $\bowtie$  is one of the relations  $<, \leq, =, >, \geq$ , and every  $v_i$  is in  $\mathcal{V}$ ) if  $\sum_i c_i \cdot A(v_i) \bowtie r$ . An assignment  $A$  satisfies a formula  $C$  if it satisfies every constraint  $c \in C$ . A formula  $C$  is integer-satisfiable or integer-consistent if there is an assignment  $A$  that satisfies  $C$ . Otherwise, it is called integer-unsatisfiable or integer-inconsistent.

A signature  $\Sigma$  consists of a set  $\Sigma^C$  of constant symbols, a set  $\Sigma^F$  of function symbols, a set  $\Sigma^P$  of predicate symbols, and a function  $ar : \Sigma^F \cup \Sigma^P \rightarrow \mathbb{N}^+$  that assigns a non-zero natural number (the arity) to every function and predicate symbol. A  $\Sigma$ -formula is a first-order logic formula constructed using the symbols in  $\Sigma$ . A  $\Sigma$ -theory  $T$  is a closed set of  $\Sigma$ -formulas (*i.e.*, contains no free variables). When  $\Sigma$  is clear from the context, we will simply write theory in place of  $\Sigma$ -theory.

**EXAMPLE 1.** Let  $\Sigma_A$  be a signature that contains a binary function *read*, a ternary function *write*, no constants, and no predicate symbols. The theory  $T_A$  of arrays (without extensionality) is defined by the following formulas [30]:

$$\forall a \forall i \forall e [\text{read}(\text{write}(a, i, e), i) = e]$$

$$\forall a \forall i \forall j \forall e [i \neq j \Rightarrow \text{read}(\text{write}(a, i, e), j) = \text{read}(a, j)].$$

A formula  $F$  is  $T$ -satisfiable or  $T$ -consistent if  $F \wedge T$  is satisfiable in the first-order sense (*i.e.*, there is an interpretation that satisfies it). A formula  $F$  is called  $T$ -unsatisfiable or  $T$ -inconsistent if it is not  $T$ -satisfiable. For formulas  $F$  and  $G$ ,  $F$   $T$ -entails  $G$  (in symbols  $F \models_T G$ ) if  $F \wedge \neg G$  is  $T$ -inconsistent.

**DEFINITION 1.** Let  $\Sigma_{\mathbb{Z}}$  be a signature that contains constant symbols  $\{0, \pm 1, \pm 2, \dots\}$ , a binary function symbol  $+$ , a unary function symbol  $-$ , and a binary predicate symbol  $\leq$ . The theory of Linear Integer Arithmetic, which we will denote by  $\mathcal{Z}$ , is the  $\Sigma_{\mathbb{Z}}$ -theory defined by the set of closed  $\Sigma_{\mathbb{Z}}$ -formulas that are true in the

standard model (an interpretation whose domain is  $\mathbb{Z}$ , in which the symbols in  $\Sigma_{\mathbb{Z}}$  are interpreted according to their standard meaning over  $\mathbb{Z}$ ).

We will use relation symbols like  $<$  that do not appear in  $\Sigma_{\mathbb{Z}}$ , and also multiplication by a constant (which is to be interpreted as repeated addition); these are only syntactic shorthands. We will frequently view an integer assignment  $A$  as the set of formulas  $\{v = A(v) \mid v \in \mathcal{V}\}$ , where  $A(v)$  is viewed as a  $\Sigma_{\mathbb{Z}}$  term. An integer assignment  $A$  viewed as a set of formulas is always  $\mathcal{Z}$ -consistent. If  $A$  is an integer assignment and  $A$  satisfies an integer linear formula  $C$ , it is also the case that  $A \models_{\mathcal{Z}} C$ . If  $A$  is an integer assignment,  $T$  is a theory and  $F$  is a formula, we will say that  $A$  is a  $T$ -model of  $F$  if  $A$  is  $T$ -consistent and  $A \models_{\mathcal{Z} \cup T} F$ .

A  $\Sigma$ -interface atom is a  $\Sigma$ -atomic formula (*i.e.*, the application of a predicate symbol or equality), possibly annotated with a variable symbol, *e.g.*,  $(x = y)^v$ . The meaning of the annotation is that the atomic formula has to hold iff the (integer) value of  $v$  is greater than 0; for our example we have  $x = y \Leftrightarrow v > 0$ .

**DEFINITION 2 (ILP Modulo  $T$  Instance).** An ILP Modulo (Theory)  $T$  instance, where the signature  $\Sigma$  of  $T$  is disjoint from  $\Sigma_{\mathbb{Z}}$ , is a triple of the form  $C, I, O$ , where:

- $C$  is an integer linear formula.
- $I$  is a set of  $\Sigma$ -interface atoms.
- $O$  is an objective function.

The variables that appear in both  $C$  and  $I$  are called interface variables.

An ILP Modulo  $T$  instance can be thought of as an integer linear program that contains terms that have meaning in  $T$ . In the above definition, the interface atoms (elements of  $I$ ) are separated from the linear constraints, *i.e.*, there are no  $\Sigma$ -terms embedded within integer linear constraints. This is not a restriction, because every quantifier-free  $(\Sigma \cup \Sigma_{\mathbb{Z}})$ -formula can be written in separate form (refer to the “Variable Abstraction” step of [26]).

**EXAMPLE 2.** Let  $\Sigma$  be a signature that contains the unary function symbol  $f$ . The formula  $f(x + 1) + f(y + 2) \geq 3$  (where  $x$  and  $y$

are variable symbols) can be written in separate form as

$$C = \{v_3 + v_4 \geq 3, v_1 = x + 1, v_2 = y + 2\}$$

$$I = \{v_3 = f(v_1), v_4 = f(v_2)\}$$

$C$  is an integer linear formula;  $I$  is a set of  $\Sigma$ -interface atoms; and  $\Sigma$  is disjoint from  $\Sigma_Z$ . Variable abstraction introduced new variables,  $v_1, \dots, v_4$ .  $C$  and  $I$  only share variable symbols.

Let  $A$  be an assignment:  $A = \{x = 2, y = 1, v_1 = 3, v_2 = 3, v_3 = 3, v_4 = 0\}$ . Clearly  $A \models_Z C$ . However,  $A \not\models_{\emptyset} I$ , where  $\emptyset$  stands for the theory of uninterpreted functions (also called the empty theory, because it has an empty set of formulas). The reason is that  $v_1 = v_2$  but  $f(v_1) \neq f(v_2)$ . In contrast, the assignment  $A' = \{x = 2, y = 1, v_1 = 3, v_2 = 3, v_3 = 3, v_4 = 3\}$  is a  $\emptyset$ -model of  $C \wedge I$ .

### 3.2 Transition System

**DEFINITION 3 (Simple Equality).** A simple equality is a constraint of the form  $v_i = c$  or  $v_i - v_j = c$ , where  $v_i$  and  $v_j$  are integer variables and  $c$  is an integer constant.

**DEFINITION 4 (Subproblem).** A subproblem is a pair of the form  $\langle C, D \rangle$ , where  $C$  is a set of constraints and  $D$  is a set of simple equalities.

A set of simple equalities resembles an assignment. Simple equalities and assignments differ, because the latter cannot contain equalities between variables with integer offsets. In a subproblem  $\langle C, D \rangle$ , we distinguish between the arbitrary constraints in  $C$  and the simpler constraints in  $D$  in order to provide a good interface for the interaction between the core ILP solver and background theory solvers that understand only a limited fragment of  $\mathcal{Z}$ . It is the responsibility of the core solver to notify the theory solver about the equalities that hold. The motivation for equalities with offsets is to capture the interaction with congruence closure algorithms that take offsets into consideration [34]. Simple equalities are a special case of integer linear constraints. Thus, we will freely conjoin (sets of) simple equalities and other constraints.

**DEFINITION 5 (State).** A state of  $BC(T)$  is a tuple  $P \parallel A$ , where  $P$  is a set of subproblems and  $A$  is either the constant **None**, or an assignment, possibly annotated with the superscript  $-\infty$ .

Our abstract framework maintains a list of open subproblems, because it is designed to allow different branching strategies. This is in contrast to an algorithm like DPLL that does not keep track of subproblems explicitly. There, subproblems are implicit, i.e., backtracking corresponds to exploring a different subproblem. ILP algorithms branch over non-Boolean variables in arbitrary ways, thus mandating explicit subproblems.

In a state  $P \parallel A$ , the assignment  $A$  is the best known ( $T$ -consistent) solution so far, if any. It has a superscript  $-\infty$  if it satisfies all the constraints, but is not optimal because the value of the objective function can be arbitrarily low. If this is the case, it is useful to provide an assignment but also to report the fact that no optimal solution exists.

The interface atoms  $I$  and the objective function  $O$  are not part of the  $BC(T)$  states because they do not change over time.  $obj(A)$  denotes the value of the objective function  $O$  under assignment  $A$ : if  $O = \sum_i c_i v_i$ , then  $obj(A) = \sum_i c_i \cdot A(v_i)$ . The objective function itself is not an argument to  $obj$  because it will be clear from the context which objective function we are referring to. For convenience, we define  $obj(\text{None}) = +\infty$  and  $obj(A^{-\infty}) = -\infty$ . Function  $lb(\langle C, D \rangle)$  returns a lower bound for the possible values of the objective function  $O$  for the subproblem  $\langle C, D \rangle$ : there is no  $A$  such that  $A$  satisfies  $C \wedge D$ , and  $obj(A) < lb(\langle C, D \rangle)$ .

Figure 8 defines the *transition relation*  $\longrightarrow$  of  $BC(T)$  (a binary relation over states). In the rules,  $c$  and  $d$  always denote integer

	$P \uplus \{\langle C, D \rangle\} \parallel A \longrightarrow$
	$P \cup \{\langle C_i, D \rangle \mid 1 \leq i \leq n\} \parallel A$
<b>Branch</b>	if $\begin{cases} n > 1 \\ D \models_Z (C \Leftrightarrow \bigvee_{1 \leq i \leq n} C_i) \\ C_i \text{ are syntactically distinct} \end{cases}$
<b>Learn</b>	$P \uplus \{\langle C, D \rangle\} \parallel A \longrightarrow P \cup \{\langle C \ c, D \rangle\} \parallel A$ if $C \wedge D \models_Z c$
<b>Forget</b>	$P \uplus \{\langle C \ c, D \rangle\} \parallel A \longrightarrow P \cup \{\langle C, D \rangle\} \parallel A$ if $C \wedge D \models_Z c$
<b>Propagate</b>	$P \uplus \{\langle C, D \rangle\} \parallel A \longrightarrow P \cup \{\langle C, D \ d \rangle\} \parallel A$ if $C \wedge D \models_Z d$
<b>Drop</b>	$P \uplus \{\langle C, D \rangle\} \parallel A \longrightarrow P \parallel A$ if $C \wedge D$ is integer-inconsistent
<b>Prune</b>	$P \uplus \{\langle C, D \rangle\} \parallel A \longrightarrow P \parallel A$ if $\begin{cases} A \neq \text{None} \\ lb(\langle C, D \rangle) \geq obj(A) \end{cases}$
<b>Retire</b>	$P \uplus \{\langle C, D \rangle\} \parallel A \longrightarrow P \parallel A'$ if $\begin{cases} A' \text{ is a } T\text{-model of } C \wedge D \wedge I \\ obj(A') < obj(A) \\ \forall B : \text{if } B \text{ is a } T\text{-model of } C \wedge D \wedge I, \\ \text{then } obj(A') \leq obj(B) \end{cases}$
<b>Unbounded</b>	$P \uplus \{\langle C, D \rangle\} \parallel A \longrightarrow \emptyset \parallel A'^{-\infty}$ if $\begin{cases} A' \text{ is a } T\text{-model of } C \wedge D \wedge I \\ obj(A') \leq obj(A) \\ \forall k : \exists B : B \text{ is a } T\text{-model of } C \wedge D \wedge I, \\ \text{then } obj(B) < k \end{cases}$
<b>T-Learn</b>	$P \uplus \{\langle C, D \rangle\} \parallel A \longrightarrow P \cup \{\langle C \ c, D \rangle\} \parallel A$ if $\exists F : C \wedge D \models_Z F, F \wedge I \models_T c$

Figure 8: The  $BC(T)$  Transition System

linear constraints and simple equalities.  $C$  (possibly subscribed) denotes an integer linear formula (set of integer linear constraints), while  $D$  denotes a set of simple equalities.  $C \ c$  stands for the set union  $C \cup \{c\}$ , under the implicit assumption that  $c \notin C$ ; similarly for  $D \ d$ .  $C$  and  $D$  are always well-formed sets, i.e., they contain no syntactically duplicate elements.  $P$  and  $P'$  stand for sets of syntactically distinct subproblems, while  $A$  and  $A'$  are integer assignments.  $P \uplus Q$  denotes the union  $P \cup Q$ , under the implicit assumption that the two sets are disjoint.  $F$  stands for a  $(\Sigma \cup \Sigma_Z)$ -formula, where  $\Sigma$  is the signature of  $T$ . The intuitive meaning of the different  $BC(T)$  rules is the following:

#### Branch

Case-split on a subproblem  $\langle C, D \rangle$ , by replacing it with two or more different subproblems  $\langle C_i, D \rangle$ . If there is a satisfying assignment for  $C \wedge D$ , this assignment will also satisfy  $C_i \wedge D$  for some  $i$ , and conversely.

### Learn, Propagate, T-Learn

Add an entailed constraint (in the case of **Learn** and **T-Learn**) or equality (**Propagate**) to a subproblem. **T-Learn** takes the theory  $T$  into account.

### Forget

Remove a constraint entailed by the remaining constraints of a subproblem.

### Drop, Prune

Eliminate a subproblem because either it is unsatisfiable (**Drop**), or it cannot lead to a solution better than the one already known.

### Retire, Unbounded

The solution to a subproblem becomes the new incumbent solution, as long as it improves upon the objective value of the previous solution. If there are solutions with arbitrarily low objective values, we don't need to consider other subproblems.

We define the binary relations  $\rightarrow^+$  and  $\rightarrow^*$  over  $\text{BC}(T)$  states as follows:  $S \rightarrow^+ S'$  if  $S \rightarrow S'$ , or there exists some state  $Q$  such that  $S \rightarrow^+ Q$  and  $Q \rightarrow S'$ .  $S \rightarrow^* S'$  if  $S = S'$  or  $S \rightarrow^+ S'$ . When convenient, we will annotate a transition arrow between two  $\text{BC}(T)$  states with the name of the rule that relates them, for example  $S \xrightarrow{\text{Branch}} S'$ .

A *starting state* for  $\text{BC}(T)$  is a state of the form  $\{\langle C, \emptyset \rangle\}$  where  $C$  is the set of integer linear constraints of an ILP Modulo  $T$  instance. A *final state* is a state of the form  $\emptyset \parallel A$  ( $A$  can also be **None**, or annotated with  $-\infty$ ). If  $\{\langle C, \emptyset \rangle\} \rightarrow^+ \emptyset \parallel \text{None}$ , then the instance is  $T$ -infeasible, while if  $\{\langle C, \emptyset \rangle\} \rightarrow^+ \emptyset \parallel A$  for  $A \neq \text{None}$ , we have found a  $T$ -model (which is optimal if an optimal solution exists).

In the subsections that follow, we provide soundness and completeness proofs for  $\text{BC}(T)$  (Theorems 1, 2, and 5). We strive to provide a general framework that can encompass a wide range of algorithms; the  $\text{BC}(T)$  transitions are thus highly non-deterministic. Subsection 3.6 provides guidelines for a practical implementation.

### 3.3 Soundness

Throughout this section, we assume an IMT instance with objective function  $O$  and a set of interface atoms  $I$ .

LEMMA 1. *For states  $P \parallel A$  and  $P' \parallel A'$  such that  $P \parallel A \rightarrow P' \parallel A'$ , if there is an assignment  $B$  such that  $B$  is a  $T$ -model of  $C \wedge D \wedge I$  for some  $\langle C, D \rangle \in P$ , then one of the following conditions holds: either (i)  $\text{obj}(A') \leq \text{obj}(B)$ , or (ii)  $B$  is a  $T$ -model of  $C' \wedge D' \wedge I$  for some  $\langle C', D' \rangle \in P'$ .*

PROOF. Assume that there is a subproblem  $p = \langle C, D \rangle \in P$  and an assignment  $B$  such that  $B$  is a  $T$ -model of  $C \wedge D \wedge I$ . If  $p \in P'$ , then (ii) holds trivially. If  $p \notin P'$ , then the transition cannot possibly be **Drop** (we cannot apply **Drop** on  $\langle C, D \rangle$  because  $B$  is an assignment that satisfies  $C \wedge D$ ). For the other rules:

- **Branch**: There are subproblems  $\langle C_1, D \rangle, \dots, \langle C_n, D \rangle$  in  $P'$  such that  $D \models_{\mathcal{Z}} (C \Leftrightarrow \bigvee_{1 \leq i \leq n} C_i)$ . Thus,  $B$  is a  $T$ -model of  $C_i \wedge D \wedge I$  for some  $i$  ( $1 \leq i \leq n$ ).  $\langle C_i, D \rangle \in P'$ , therefore (ii) holds.
- **Prune**:  $\text{obj}(A') \leq \text{lb}(C \wedge D) \leq \text{obj}(B)$ . Condition (i) holds.
- **Retire, Unbounded**:  $\text{obj}(A') \leq \text{obj}(B)$ , therefore (i) holds.
- **Learn, Forget, Propagate, T-Learn**: there is a subproblem  $\langle C', D' \rangle \in P'$  such that  $C \wedge D \wedge I \models_{\mathcal{Z} \cup T} C' \wedge D'$ , and therefore  $B$  is a  $T$ -model of  $C' \wedge D' \wedge I$ .

LEMMA 2. *For states  $P \parallel A$  and  $P' \parallel A'$  such that  $P \parallel A \rightarrow P' \parallel A'$ ,  $\text{obj}(A') \leq \text{obj}(A)$ .*

PROOF. The only rules that modify the assignment are **Retire** and **Unbounded**; the conditions under which we can apply them imply  $\text{obj}(A') \leq \text{obj}(A)$ . For any other rule,  $A = A'$ .

LEMMA 3. *For states  $P \parallel A$  and  $P' \parallel A'$  such that  $P \parallel A \rightarrow P' \parallel A'$ , if  $A \neq A'$  then  $A'$  is a  $T$ -model of  $C \wedge D \wedge I$  for some  $\langle C, D \rangle \in P$ .*

PROOF. The conditions of **Retire** and **Unbounded** guarantee that  $A'$  is a  $T$ -model of  $C \wedge D \wedge I$  for some  $\langle C, D \rangle \in P$ . No other rule modifies the assignment.

LEMMA 4. *For states  $P \parallel A$  and  $P' \parallel A'$  such that  $P \parallel A \rightarrow^* P' \parallel A'$ , if there is an assignment  $B$  such that  $B$  is a  $T$ -model of  $C \wedge D \wedge I$  for some  $\langle C, D \rangle \in P$ , then one of the following conditions holds:  $\text{obj}(A') \leq \text{obj}(B)$ , or  $B$  is a  $T$ -model of  $C' \wedge D' \wedge I$  for some  $\langle C', D' \rangle \in P'$ .*

PROOF. We induct on the length  $n$  of the sequence of transitions.

- Induction base:  $n = 0$ .  $P \parallel A = P' \parallel A'$ ; obvious.
- Induction step: assume that the property holds for any sequence of  $n - 1$  transitions, where  $n \geq 1$ . We will prove that it holds for any sequence of transitions  $P_0 \parallel A_0 \rightarrow \dots \rightarrow P_{n-1} \parallel A_{n-1} \rightarrow P_n \parallel A_n$ . Assume there is an assignment  $B$  such that  $B$  is a  $T$ -model of  $C_0 \wedge D_0 \wedge I$  for some  $\langle C_0, D_0 \rangle \in P_0$ . By the induction hypothesis, one of the two following conditions holds:
  - $\text{obj}(A_{n-1}) \leq \text{obj}(B)$ : then  $\text{obj}(A_n) \leq \text{obj}(A_{n-1}) \leq \text{obj}(B)$  from lemma 2.
  - $B$  is a  $T$ -model of  $C_{n-1} \wedge D_{n-1} \wedge I$  for some subproblem  $\langle C_{n-1}, D_{n-1} \rangle \in P_{n-1}$ : our proof obligation follows from lemma 1 applied to the transition  $P_{n-1} \parallel A_{n-1} \rightarrow P_n \parallel A_n$ .

LEMMA 5. *For states  $P \parallel A$  and  $P' \parallel A'$  such that  $P \parallel A \rightarrow P' \parallel A'$ ,  $\bigvee_{\langle C, D \rangle \in P'} C \wedge D \models_{\mathcal{Z}} \bigvee_{\langle C, D \rangle \in P} C \wedge D$ .*

PROOF. We case-split on the  $\text{BC}(T)$  transitions.

- The rules **Prune**, **Drop**, and **Retire** can only make the disjunction of the subproblems in  $P'$  stronger, because a subproblem is eliminated and the rest of the subproblems remain intact.
- For **Unbounded**,  $\bigvee_{\langle C, D \rangle \in P'} C \wedge D \equiv \text{false}$ ;  $\text{false} \models_{\mathcal{Z}} \bigvee_{\langle C, D \rangle \in P} C \wedge D$ .
- The rules **Learn**, **Forget**, and **Propagate** and substitute a subproblem for a  $\mathcal{Z}$ -equivalent one.
- The rule **T-Learn** adds a constraint to a subproblem, and therefore makes the disjunction in  $P'$  stronger.
- The rule **Branch** replaces a subproblem  $\langle C, D \rangle$  with a set of subproblems whose disjunction is  $\mathcal{Z}$ -equivalent to  $C \wedge D$ .

LEMMA 6. *For states  $P \parallel A$  and  $P' \parallel A'$  such that  $P \parallel A \rightarrow^* P' \parallel A'$ ,  $\bigvee_{\langle C, D \rangle \in P'} C \wedge D \models_{\mathcal{Z}} \bigvee_{\langle C, D \rangle \in P} C \wedge D$ .*

PROOF. Induction on the length of the sequence of transitions and application of lemma 5.

THEOREM 1. *For a formula  $C$ , if  $\{\langle C, \emptyset \rangle\} \parallel \text{None} \rightarrow^* \emptyset \parallel \text{None}$ , then  $C \wedge I$  is  $T$ -unsatisfiable.*

PROOF. Assume that there is an assignment  $A$  such that  $A$  is a  $T$ -model of  $C$ . Then, from lemma 4 either there exists  $\langle C', D' \rangle \in \emptyset$  such that  $A$  is a  $T$ -model of  $C' \wedge D' \wedge I$  (which cannot possibly be true), or  $+\infty = \text{obj}(\text{None}) \leq \text{obj}(A)$  (contradiction, because  $\text{obj}(A)$  has to be finite).

THEOREM 2. *For a formula  $C$  and an assignment  $A$ , if*

$$\{\langle C, \emptyset \rangle\} \parallel \text{None} \rightarrow^* \emptyset \parallel A$$

where  $A \neq \text{None}$ , then (a)  $A$  is a  $T$ -model of  $C \wedge I$ , and (b) there is no assignment  $B$  such that  $B$  is a  $T$ -model of  $C \wedge I$  and  $\text{obj}(B) < \text{obj}(A)$ .

PROOF. (a) The sequence of transitions from  $\{\langle C, \emptyset \rangle\} \parallel \text{None}$  to  $\emptyset \parallel A$  has to be of the following form:

$$\begin{array}{ccc} \{\langle C, \emptyset \rangle\} \parallel \text{None} & \xrightarrow{*} S_1 \parallel A_1 & \xrightarrow{\text{Retire/Unbounded}} \\ & & S \parallel A \xrightarrow{*} \emptyset \parallel A \end{array}$$

There is at least one **Retire** or **Unbounded** step, as these are the only rules that can modify the assignment. Consider the last such step. The conditions on **Retire** and **Unbounded** steps require that  $A$  is a  $T$ -model of  $I \wedge \bigvee_{\langle C, D \rangle \in S_1} (C \wedge D)$ . From lemma 6,  $\bigvee_{\langle C, D \rangle \in S_1} (C \wedge D) \models_{\mathcal{Z}} C$ . Therefore,  $A$  is a  $T$ -model of  $C$ .

(b) Follows from lemma 4.

### 3.4 Termination

An algorithm that implements the transition system in Figure 8 has flexibility with respect to branching: in particular, a non-terminating branching strategy is possible. In addition, an infinite number of ILP inequalities is implied by any arbitrary set of ILP constraints, and they can be added to the problem by means of the **Learn** rule, or forgotten by means of **Forget**. To ensure termination, we impose constraints on how the rules **Branch**, **Learn**, **Forget**, and **T-Learn** can be applied.

**THEOREM 3 (Termination).** *A sequence of  $BC(T)$  transitions is finite if and only if the following conditions hold:*

- (a) *The sequence has a finite number of **Branch** steps.*
- (b) *Every infinite suffix of the sequence includes a step that is not one of **Learn**, **Forget**, and **T-Learn**.*

PROOF. The only if direction is obvious. Suppose a sequence  $\sigma$  satisfies (a) and (b). Then, by (a), it has a suffix  $\tau$  with no **Branch** steps. The number of subproblems in  $\tau$  never increases (that would require a **Branch** step), and decreases whenever we perform a **Drop**, **Prune**, **Retire**, or **Unbounded** step. Thus,  $\tau$  can only contain a finite number of **Drop**, **Prune**, **Retire**, and **Unbounded** steps. Therefore,  $\tau$  has a suffix,  $\xi$ , without any **Drop**, **Prune**, **Retire**, and **Unbounded** steps. Now  $\xi$  can only contain a finite number of **Propagate** steps, because we can learn at most one simple equality for each pair of variables in a subproblem. So  $\xi$  has a suffix,  $\zeta$ , with only **Learn**, **Forget**, and **T-Learn** steps, but by condition (b)  $\zeta$  has to be finite. Therefore  $\sigma$  is finite.

### 3.5 Completeness

**DEFINITION 6 (Stably Infinite Theory).** *A  $\Sigma$ -theory  $T$  is called stably infinite if for every  $T$ -satisfiable quantifier-free  $\Sigma$ -formula  $F$  there exists an interpretation satisfying  $F \wedge T$  whose domain is infinite.*

**DEFINITION 7 (Arrangement).** *Let  $E$  be an equivalence relation over a set of variables  $V$ . The set*

$$\begin{aligned} \alpha(V, E) = & \{x = y \mid xEy\} \cup \\ & \{x \neq y \mid x, y \in V \text{ and not } xEy\} \end{aligned}$$

is the arrangement of  $V$  induced by  $E$ .

Note that  $\mathcal{Z}$  is a stably infinite theory. We build upon the following result on the combination of signature-disjoint stably infinite theories:

**THEOREM 4 (Combination of Stably Infinite Theories [26]).** *Let  $T_i$  be a stably infinite  $\Sigma_i$ -theory, for  $i = 1, 2$ , and let  $\Sigma_1 \cap \Sigma_2 = \emptyset$ .*

*Also, let  $\Gamma_i$  be a conjunction of  $\Sigma_i$  literals.  $\Gamma_1 \cup \Gamma_2$  is  $(T_1 \cup T_2)$ -satisfiable iff there exists an equivalence relation  $E$  of the variables shared by  $\Gamma_1$  and  $\Gamma_2$  such that  $\Gamma_i \cup \alpha(V, E)$  is  $T_i$ -satisfiable, for  $i = 1, 2$ .*

The theorem above implies that the combination of  $T_1 = \mathcal{Z}$  and another stably infinite theory is decidable: we can pick an arrangement over the variables shared by the two sets of literals non-deterministically and perform two  $T_i$ -satisfiability checks. We show how  $BC(T)$  can meet the hypotheses of the theorem to ensure completeness.

**THEOREM 5 (Completeness).**  *$BC(T)$  provides a complete optimization procedure for the ILP Modulo  $T$  problem, where  $T$  is a decidable stably infinite theory.*

PROOF SKETCH. Let  $\mathcal{C}, I, O$  be an ILP Modulo  $T$  instance. Assume that

$$\langle \mathcal{C}, \emptyset \rangle \parallel \text{None} \xrightarrow{*} P \parallel A,$$

and that for every  $\langle C, D \rangle \in P$  the following conditions hold: (a) there is an equivalence relation  $E_{\langle C, D \rangle}$  over the set of interface variables  $V$  of the ILP Modulo  $T$  instance, such that  $C \wedge D \models_{\mathcal{Z}} \alpha(V, E_{\langle C, D \rangle})$ , and (b)  $C \wedge D \models_{\mathcal{Z}} v > 0$ , or  $C \wedge D \models_{\mathcal{Z}} v \leq 0$  for every  $v$  that appears as the annotation of an interface atom in  $I$ . Then we can solve the IMT instance to optimality as follows. For every subproblem  $\langle C, D \rangle \in P$ , we decide the  $T$  instance:

$$\begin{aligned} & \{t \mid t \in I \text{ and } t \text{ is not annotated}\} \cup \\ & \{t \mid t^v \in I \text{ and } C \wedge D \models_{\mathcal{Z}} v > 0\} \cup \\ & \{\neg t \mid t^v \in I \text{ and } C \wedge D \models_{\mathcal{Z}} v \leq 0\} \cup \\ & \alpha(V, E_{\langle C, D \rangle}) \end{aligned}$$

If the instance is  $T$ -unsatisfiable, then  $C \wedge D \wedge I$  is  $T$ -unsatisfiable. If it is  $T$ -satisfiable, any integer solution for  $C \wedge D$  will be a  $T$ -model. For the  $T$ -unsatisfiable subproblems, we apply **T-Learn** to learn an integer-infeasible constraint (always possible because a  $T$ -inconsistent formula implies anything, e.g.,  $0 < 0$ ) and subsequently apply **Drop**. If all the subproblems are  $T$ -unsatisfiable, we reach a final state  $\emptyset \parallel \text{None}$ . If there are  $T$ -satisfiable subproblems, it suffices to apply a complete algorithm for obtaining integer solutions (e.g., Gomory's method [17]) as we have already established  $T$ -consistency. The basic steps of such algorithms (for instance, adding a cutting plane) can be described by means of  $BC(T)$  steps. If the  $BC(T)$  sequence adheres to the hypotheses of Theorem 3, it will be finite. Also, note that objective functions do not hinder completeness: it suffices to recognize an unbounded subproblem [7] and apply **Unbounded**.

It is easy for an implementation of  $BC(T)$  to guarantee that after a finite number of steps every subproblem will entail an arrangement of the interface variables. For every pair of interface variables  $x$  and  $y$  and every subproblem, we apply the **Branch** rule to obtain three new subproblems each of which contains one of the constraints  $x < y$ ,  $x = y$ , and  $x > y$ . Similarly, we can branch to obtain a truth value for  $v > 0$  for every  $v$  that appears as the annotation of an interface atom.

### 3.6 Discussion

A straightforward implementation of  $BC(T)$  is a branch-and-cut algorithm for ILP (branching and learning entailed constraints with cutting plane techniques) that communicates with theory solvers.

Practical branch-and-cut algorithms rely heavily on continuous relaxations of the subproblems. Although  $BC(T)$  does not explicitly make use of relaxations, it is designed to facilitate their use: (a) The lower-bounding procedure *lb* can rely on solutions to continuous relaxations, as the best integral solution can be at most



as good as the best non-integral solution. (b) Solutions to relaxations can guide the use of the **Branch** rule: *e.g.*, if a solution assigns a non-integer value  $r$  to variable  $v$ , we can branch on whether  $v \geq \lceil r \rceil$  or  $v \leq \lfloor r \rfloor$ . (c) If the relaxation of a subproblem is infeasible, then the subproblem itself is infeasible; we apply the **Drop** rule. (d) If the relaxation of a subproblem has an  $T$ -consistent integer solution, we can apply **Retire**.

The  $BC(T)$  architecture does not preclude specialized techniques for special classes of linear constraints. For example, an implementation can use the two-watched-literal scheme to accelerate Boolean Constraint Propagation on clauses. Our goal is not to replace propositional reasoning, but rather to shift a broader class of constraints from background theories to the core solver.

#### 4. SMT as IMT

IMT can be thought of as an extension to SMT, *i.e.*, an IMT solver can be used in place of an SMT solver. To support this claim, we show how to handle propositional structure within IMT, and thus solve SAT Modulo  $\mathcal{Z}$  problems. We also prove that a version of the  $BC(T)$  transition system can simulate  $DPLL(T)$  [36]. A simulation in the reverse direction is impossible, because  $BC(T)$  natively handles optimization, while  $DPLL(T)$  does not.

##### 4.1 Handling Propositional Structure

Clauses can be thought of as a special case of linear constraints: *i.e.*, a clause  $\forall_i l_i$  is equivalent to  $\sum_i l_i \geq 1$  (for translating a clause to a linear expression, a negative literal  $\neg v_i$  appears as  $1 - v_i$  while a positive literal remains intact). Thus, we can translate the propositional structure in an SMT instance as an equivalent set of linear constraints.

SAT Modulo  $\mathcal{Z}$  instances contain  $\mathcal{Z}$  theory atoms under the propositional connectives. For every such atom  $t$ , we define an integer variable  $v(t)$  such that  $v(t) \Leftrightarrow t$ . It suffices to show how this works for linear inequalities of the form  $t = (\sum_i c_i \cdot v_i \leq r)$ ; other relations can be expressed in terms of  $\leq$  and the propositional connectives. Assume that there are constants  $m, k$  such that  $m < \sum_i c_i \cdot v_i \leq k$  always holds. There exist such  $m, k$  if all variables are bounded (we can always impose bounds that preserve satisfiability [40]). The direction  $v(t) \Rightarrow t$  can be expressed as  $\sum_i c_i \cdot v_i \leq r + (k - r) \cdot (1 - v(t))$ ; for the opposite direction we have  $\sum_i c_i \cdot v_i > r + (m - r) \cdot v(t)$ . Subsequently  $v(t)$  appears in clauses in place of  $t$ .

The technique we just described relies on computing variable bounds which can be of cosmic magnitude, but this is only a minor concern. In the worst case, large bounds will lead to weak continuous relaxations; the relaxations will become stronger once we start branching on the Boolean variables. Modern ILP solvers provide an alternative called *indicator constraints*<sup>2</sup>, *i.e.*, natively supported constraints of the form  $v = 1 \Rightarrow \sum_i c_i \cdot v_i \leq r$ , where  $v$  is a  $\{0, 1\}$  variable.  $BC(T)$  does not explicitly provide indicator constraints, in order to stay within the standard formulation of ILP.

##### 4.2 Simulating $DPLL(T)$

A  $DPLL(T)$  assignment is a sequence of literals that does not contain both a literal and its negation. Some literals are annotated as decision literals (written as  $l^d$ ). A  $DPLL(T)$  state is either the special state **FailState**, or a tuple of the form  $M \parallel F$ , where  $M$  is a  $DPLL(T)$  assignment and  $F$  is a propositional formula (set of clauses). For precise definitions and transition rules refer to [36]. To avoid confusion with  $BC(T)$ , we denote the  $DPLL(T)$  transition relation by  $\rightarrow_D$ .

We add an extra rule (called **Subsume**) to the  $BC(T)$  transition system. We refer to the resulting transition system as  $BC(T)^+$ , and

denote the  $BC(T)^+$  transition relation by  $\rightarrow_I$ .

$$\text{Subsume} \quad \begin{array}{l} P \uplus \{\langle C, D \rangle, \langle C', D' \rangle\} \parallel A \longrightarrow \\ P \cup \{\langle C, D \rangle\} \parallel A \end{array}$$

$$\text{if } C' \wedge D' \models_{\mathcal{Z}} C \wedge D$$

Intuitively, **Subsume** drops a subproblem  $\langle C', D' \rangle$  if there is a more general subproblem  $\langle C, D \rangle$ , *e.g.*, if  $\langle C', D' \rangle$  was derived from  $\langle C, D \rangle$  by making some decisions. **Subsume** is not part of the  $BC(T)$  system as described in Section 3, because we do not have evidence that such a rule would be useful for branch-and-cut algorithms. The sole purpose of **Subsume** is to help us simulate the  $DPLL(T)$  *T*-Backjump and **Restart** rules. We would not need **Subsume** if we were to simulate a version of  $DPLL(T)$  with a backjumping rule more restrictive than *T*-Backjump (*e.g.*, jumping right above the highest *T*-inconsistent decision level), and without a **Restart** rule.

The soundness proof of Section 3 can easily be adapted for  $BC(T)^+$ . Uncontrolled **Subsume** poses an extra threat for termination; the hypotheses of Theorem 3 have to be strengthened accordingly.

DEFINITION 8 (Simulation Function  $R$ ).

$$R(\text{FailState}) = \emptyset \parallel \text{None}$$

$$R(M \parallel F) = \{\langle \llbracket F \rrbracket, \llbracket M \rrbracket \rangle\} \cup L(M, F) \parallel \text{None},$$

where

$$L(M, F) = \langle \llbracket F \rrbracket, \emptyset \rangle \cup$$

$$\{\langle \llbracket F \rrbracket, \llbracket N \rrbracket \rangle \mid \exists l : N \text{ } l^d \text{ prefix of } M\}$$

By  $\llbracket \cdot \rrbracket$  we denote the translation from a  $DPLL(T)$  clause (or formula) to a  $BC(T)$  linear inequality (or set thereof). We have outlined how the translation happens in Subsection 4.1. A positive  $DPLL(T)$  literal viewed as a simple equality is of the form  $v = 1$  (for a negative literal we have  $v = 0$ ). The basic idea behind the simulation function  $R$  is that the negation of each  $DPLL(T)$  decision is like a subproblem yet to be explored.  $DPLL(T)$  does not explicitly represent subproblems, as they can be easily reconstructed from the assignment.

THEOREM 6 (Simulation). *For  $DPLL(T)$  states  $S$  and  $S'$  such that  $S \rightarrow_D S'$ ,  $R(S) \rightarrow_I^+ R(S')$ .*

PROOF. We case-split on the  $DPLL(T)$  transition rules.

**Decide:**  $S = M \parallel F$  and  $S' = M \parallel l \parallel F$  where  $M$  is an assignment,  $F$  is a formula, and  $l$  is a literal such that  $l$  or  $\neg l$  occurs in  $F$ , and  $l$  is undefined in  $M$ .

$$R(S) = \{\langle \llbracket F \rrbracket, \llbracket M \rrbracket \rangle\} \cup L(M, F) \parallel \text{None}$$

$$\xrightarrow[\text{Branch}]{} \{\langle \llbracket F \rrbracket \wedge \llbracket l \rrbracket, \llbracket M \rrbracket \rangle, \langle \llbracket F \rrbracket, \llbracket M \rrbracket \rangle\} \cup L(M, F) \parallel \text{None}$$

$$\xrightarrow[\text{Propagate}]{} \{\langle \llbracket F \rrbracket \wedge \llbracket l \rrbracket, \llbracket M \rrbracket \wedge \llbracket l \rrbracket \rangle, \langle \llbracket F \rrbracket, \llbracket M \rrbracket \rangle\} \cup L(M, F) \parallel \text{None}$$

$$\xrightarrow[\text{Forget}]{} \{\langle \llbracket F \rrbracket, \llbracket M \rrbracket \wedge \llbracket l \rrbracket \rangle, \langle \llbracket F \rrbracket, \llbracket M \rrbracket \rangle\} \cup L(M, F) \parallel \text{None}$$

$$= \{\langle \llbracket F \rrbracket, \llbracket M \rrbracket l^d \rangle, \langle \llbracket F \rrbracket, \llbracket M \rrbracket \rangle\} \cup L(M, F) \parallel \text{None}$$

$$= \{\langle \llbracket F \rrbracket, \llbracket M \rrbracket l^d \rangle\} \cup L(M \parallel l^d, F) \parallel \text{None}$$

$$= R(S')$$

**Fail:**  $S = M \parallel F, C$  and  $S' = \text{FailState}$ , where  $M \models \neg C$  and  $M$  contains no decision literals.  $L(M, (F, C)) = \emptyset$ .  $\llbracket F, C \rrbracket \wedge \llbracket M \rrbracket$  is  $\mathcal{Z}$ -inconsistent, because  $M \models \neg C$  (where  $\models$  denotes  $DPLL(T)$  propositional entailment).

$$R(S) = \{\langle \llbracket F, C \rrbracket, \llbracket M \rrbracket \rangle\} \parallel \text{None} \xrightarrow[\text{Prune}]{} \emptyset \parallel \text{None} = R(S')$$

<sup>2</sup><http://j.mp/NdkZZ1> (CPLEX); <http://j.mp/Nd1mDs> (SCIP)

**UnitPropagate:**  $S = M \parallel F, C \vee l$  and  $S' = M \parallel F, C \vee l$  where  $M \models \neg C$  and  $l$  is undefined in  $M$ . Notice that  $L(M, F) = L((M \parallel l), F)$ .

$$\begin{aligned} R(S) &= \{ \langle [F, C \vee l], [M] \rangle \} \cup L(M, F) \parallel \text{None} \\ &\xrightarrow[\text{Propagate}]{I} \{ \langle [F, C \vee l], [M \parallel l] \rangle \} \cup L(M, F) \parallel \text{None} \\ &= R(S') \end{aligned}$$

**T-Propagate:**  $S = M \parallel F$ , and  $S' = M \parallel F$  where  $M \models_T l$ ,  $l$  or  $\neg l$  occurs in  $F$ , and  $l$  is undefined in  $M$ . Notice that  $\llbracket F \rrbracket, \llbracket M \rrbracket \models_Z \llbracket M \rrbracket$  and  $\llbracket M \rrbracket, I \models_T \llbracket l \rrbracket$ .

$$\begin{aligned} R(S) &= \{ \langle [F], [M] \rangle \} \cup L(M, F) \parallel \text{None} \\ &\xrightarrow[T\text{-Learn}]{I} \{ \langle [F] \wedge [l], [M] \rangle \} \cup L(M, F) \parallel \text{None} \\ &\xrightarrow[\text{Propagate}]{I} \{ \langle [F] \wedge [l], [M \parallel l] \rangle \} \cup L(M, F) \parallel \text{None} \\ &\xrightarrow[\text{Forget}]{I} \{ \langle [F], [M \parallel l] \rangle \} \cup L(M, F) \parallel \text{None} \\ &= R(S') \end{aligned}$$

**T-Backjump:**  $S = M \parallel^d N \parallel F, C$  and  $S' = M \parallel^d N \parallel F, C$  where  $M \parallel^d N \models \neg C$ , and there is some clause  $C' \vee l'$  such that  $F, C \models_T C' \vee l'$  and  $M \models \neg C'$ ,  $l'$  is undefined in  $M$ , and  $l'$  or  $\neg l'$  occurs in  $F$  or in  $M \parallel^d N$ .

$$\begin{aligned} R(S) &= \{ \langle [F, C], [M \parallel^d N] \rangle \} \cup L((M \parallel^d N), (F, C)) \parallel \text{None} \\ &\xrightarrow[\text{Drop}]{I} L((M \parallel^d N), (F, C)) \parallel \text{None} \\ &\xrightarrow[\text{Subsume}]{I} \{ \langle [F, C], [M] \rangle \} \cup L(M, (F, C)) \parallel \text{None} \\ &\xrightarrow[T\text{-Learn}]{I} \{ \langle [F, C, l'], [M] \rangle \} \cup L(M, (F, C)) \parallel \text{None} \\ &\xrightarrow[\text{Propagate}]{I} \{ \langle [F, C, l'], [M \parallel l'] \rangle \} \cup L(M, (F, C)) \parallel \text{None} \\ &\xrightarrow[\text{Forget}]{I} \{ \langle [F, C], [M \parallel l'] \rangle \} \cup L(M, (F, C)) \parallel \text{None} \\ &= \{ \langle [F, C], [M \parallel l'] \rangle \} \cup L((M \parallel l'), (F, C)) \parallel \text{None} \\ &= R(S') \end{aligned}$$

**T-Learn:**  $S = M \parallel F$  and  $S' = M \parallel F, C$ . All subproblems in  $R(S)$  have the same linear constraints,  $\llbracket F \rrbracket$ ; we can apply the *T-Learn* rule to each subproblem and learn  $\llbracket C \rrbracket$ . Thus,  $R(S) \xrightarrow[\text{Learn}]{+} R(S')$ .

**T-Forget:** similar to *T-Learn*:  $R(S) \xrightarrow[\text{Forget}]{+} R(S')$ .

**T-Restart:**  $S = M \parallel F$  and  $S' = \emptyset \parallel F$ .

$$\begin{aligned} R(S) &= \{ \langle [F], [M] \rangle \} \cup L(M, F) \parallel \text{None} \\ &\xrightarrow[\text{Subsume}]{I} \{ \langle [F], \emptyset \rangle \} \parallel \text{None} \\ &= R(S') \end{aligned}$$

The simulation sometimes performs multiple  $\text{BC}(T)$  transitions for a single  $\text{DPLL}(T)$  transition. This is because we have to perform a certain action for multiple  $\text{BC}(T)$  subproblems (as many as the number of decision literals in the  $\text{DPLL}(T)$  assignment). To put Theorem 6 into perspective,  $\text{BC}(T)$  subproblems are an abstract construct, and do not necessarily resemble the data structures used by a practical implementation that branches in a controlled way, like  $\text{DPLL}(T)$ . Conversely, the number of steps that the simulation requires is not to be interpreted as indicative of the performance of an IMT solver.

## 5. Experiments

As we described in Section 2.1.3, our approach for synthesizing architectural models combines an ILP solver (which handles

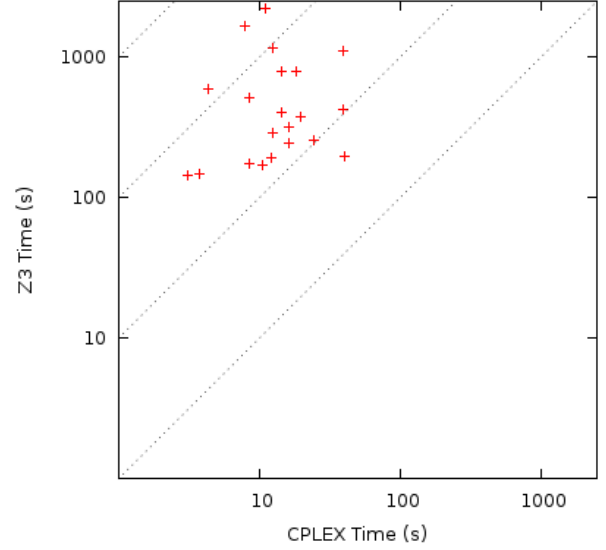


Figure 9: Z3 (best custom encoding) against CPLEX (no encoding)

structural and resource constraints) with a decision procedure for scheduling. We now provide experimental evidence that an ILP solver is essential for scalability, *i.e.*, a SAT-centric combination of decision procedures (SMT) would not work.

To evaluate SMT against our resource and structural requirements, we perform the following experiment: we run CoBaSA on a characteristic 787-derived instance and record all ILP solver queries (21 in total). The instances are essentially pseudo-Boolean (PB) problems, *i.e.*, they consist of linear constraints over Boolean (or  $\{0, 1\}$ ) variables. The ILP solver we use is CPLEX, a state-of-the-art industrial solver. The SMT solver we use is Z3 [10], the QF\_LIA (Quantifier-Free Linear Integer Arithmetic) winner of SMT-COMP 2011.<sup>3</sup>

In our first experiment, we compare CPLEX with Z3, using the obvious encoding: we generate conjunctions of linear constraints, using integer variables bounded from 0 to 1. CPLEX is given the constraints using its format and Z3 is given constraints in the SMT-LIB 2.0 format. With a timeout of 10 hours, Z3 fails to provide a solution for *any* of the instances. In contrast, CPLEX solves all instances, requiring an average of 16 seconds per instance. Therefore, CPLEX outperforms Z3 by at least four orders of magnitude.

In our next experiment, we tried numerous encodings to see if we could trade human time for improved Z3 performance. We report on the best we were able to accomplish. Observe that the variables in the above instances are in fact Boolean. Some of the linear constraints are clauses, *i.e.*, of the form  $\sum l_i \geq 1$  for literals  $l_i$ . It makes sense to help Z3 by encoding such constraints as clauses. To do this, we declare all variables to be Boolean. Since almost all variables also appear in arithmetic contexts where they are multiplied by constants greater than 1, we translate such constraints as demonstrated by the following example: the linear constraint  $x + y + 2 \cdot z \geq 2$  becomes  $(\geq (+ (ite x 1 0) (ite y 1 0) (ite z 2 0)) 2)$ . With this encoding, Z3 solves all problems, but performance is 36 times slower than CPLEX on average. Figure 9 visualizes the performance of Z3, using the best encoding we could find, versus

<sup>3</sup> See <http://www.smt-comp.org>. In fact, Z3 won most other categories. It also outperformed all other solvers in all arithmetic categories of SMT-COMP 2012, although the previous winner does not officially compete.

CPLEX, using the default encoding (for which Z3 provided no solutions). Notice that even with human intervention, Z3 is between 1 and 2 orders of magnitude slower than CPLEX for most of the problems.

Our last SAT-based option is to reduce the structural and architectural requirements to propositional SAT, and bypass SMT and QF\_LIA altogether. In essence, we have to translate PB constraints to SAT. We explored this option in depth [27]. In summary, we did manage to approach the performance of ILP, but it required a custom PB solver that combines incremental translation to SAT with reasoning at the PB level; eager translation followed by SAT solving was orders of magnitude worse. For the sequence of 21 instances, our PB solver is four times slower than CPLEX.

So far our experimentation has focused on feasibility problems, while in reality commercial ILP solvers like CPLEX are primarily developed for optimization. There is no wide consensus in the SAT and SMT community on how to best perform optimization. Relatively simple techniques are applied, *e.g.*, decrement-by-one. As we argued previously, the use of objective functions is of fundamental importance to synthesis problems. When we also consider optimization, this tends to significantly amplify the performance gap between ILP and SAT-based solvers (in favor of ILP solvers). To highlight this, we perform one last experiment. We encode the *fundraising* problem in CoBaSA: 56 guests and their spouses (78 people in total) are invited to a dinner. Some of the guests are Northeastern University (NEU) officials, while the rest are supporters of the university. The guests have to be mapped to 8 tables of 10 seats. We obviously have resource constraints for seats. Each guest consumes one seat, unless they bring along their spouse, in which case they consume two seats. There are separation constraints, because some supporters dislike each other. The objective is to maximize the number of supporters co-located with NEU officials (the latter are planning to ask for contributions). The problem is moderately hard, as it takes 9 seconds for CPLEX to find an optimal allocation. Our PB solver finds a feasible allocation in less than a second; however it takes close to two hours to find a provably optimal solution.

Finally, it is worth pointing out that evaluation of ILP solvers against SAT-inspired engines with respect to optimization takes place yearly, as part of the PB Competition.<sup>4</sup> There, CPLEX and SCIP [4] dominate the SAT-based algorithms.

## 6. Related Work

### 6.1 Nelson-Oppen, SAT, and SMT

The seminal work of Nelson and Oppen [32] provided the foundations for combining decision procedures for different theories. Tinelli and Harandi [41] revisit the Nelson-Oppen method and propose a non-deterministic variant that works for non-convex stably infinite theories. Manna and Zarba provide a detailed survey of Nelson-Oppen and related methods [26].

ILP Modulo Theories resembles Satisfiability Modulo Theories, with ILP as the core formalism instead of SAT. SMT has been the subject of active research over the last decade (*e.g.*, [5, 11, 36]). Nieuwenhuis, Oliveras and Tinelli [36] present DPLL( $T$ ), an abstract framework that can be used to formally reason about the lazy SMT technique. Based on DPLL( $T$ ), one can obtain a solver for satisfiability modulo a theory  $T$  by simply plugging in the corresponding theory solver  $\text{Solver}_T$ . The SMT framework has been extended to deal with optimization problems [8, 35, 39].

Different fragments of Linear Arithmetic have been studied as background theories for the SMT framework. Dutertre and de Moura [12] present a Simplex-based Linear Arithmetic solver for

DPLL( $T$ ). The same authors employ a branch-and-cut strategy to handle integer or mixed integer problems [13]. Overall, Linear Arithmetic as a background theory for SMT differs significantly from IMT, because in our case the *core* solver is an ILP solver.

A family of linear solvers that apply ideas from SAT has recently appeared [21, 24, 38]. These algorithms essentially generalize DPLL-style search for a satisfying assignment to non-Boolean variables, and can be thought of as steps towards SMT with a non-propositional core engine. Our research direction is complementary, as we do not focus on the core solver, but rather provide a general framework for combining ILP with theories.

### 6.2 Linear Programming

Branch-and-cut algorithms [31] combine branching (branch-and-bound) with cutting plane techniques: adding violated inequalities (cuts) to the linear formulation. Different methods have been studied for generating cuts for general integer programming problems, starting with the seminal work of Gomory [17]. Cuts can also be generated in a problem-specific way, *e.g.*, for TSP [18]. Problem-specific cuts are analogous to theory lemmas in our framework.

Another family of linear programming techniques that bears resemblance to IMT is Benders decomposition [6]. The linear programming problem is split into a master problem (which has only a subset of the original variables) and a subproblem. The master problem is solved first, and then the subproblem is solved with the values of the master problem fixed (“trial” values). If the “trial” values are unacceptable, a cut is derived and added to the master problem. Logic-based Benders Decomposition [20] generalizes the strategy so that the master problem, the subproblem, or both are not necessarily linear programs. In IMT, the problem is “decomposed” into a core ILP instance and background theory problems.

Linear and integer programming solvers generally perform floating-point (FP) and thus inexact calculations. Faure et al. experiment with the inexact CPLEX solver as a theory solver [15] and observe wrong answers. For many applications, numerical inaccuracies are not a concern, *e.g.*, the noise in the model overshadows the floating point error intervals, or an answer close enough to the theoretical optimal suffices. However, accuracy is often critical. Recent work [9, 33] proposes using FP arithmetic as much as possible (especially for solving continuous relaxations), while ensuring that cutting planes, infeasibility certificates, and bounds obtained from relaxations are safe. IMT solvers can be built on top of both exact and inexact solvers, depending on the requirements.

### 6.3 Languages

We are not the first to pursue the idea of an extensible language. Lisp and Scheme implementations have long provided powerful macro systems [14, 23, among many others]. The modern-day Racket platform provides an arsenal of extension mechanisms [42] for defining DSLs and hosting different paradigms. CoBaSA is a specification language as opposed to a programming language; thus, the ways to achieve extensibility differ, *e.g.*, we need infrastructure for combining decision procedures.

Modeling a problem directly as a linear program can be difficult and error-prone. Higher-level modeling languages like AMPL [37], GAMS [3], and Zimpl [22] aim at making optimization technology more accessible, an objective that they share with CoBaSA. Unlike these languages, we allow the programmer to extend the underlying formalism by means of theories, and expose theories to the high-level language.

## 7. Conclusions and Future Work

We introduced the ILP Modulo Theories (IMT) framework for describing problems that consist of linear constraints along with back-

<sup>4</sup> See <http://www.cril.univ-artois.fr/PB12/>

ground theory constraints. We did this via the abstract  $BC(T)$  transition system that captures the essence of the branch-and-cut family of algorithms for solving IMT problems. We showed that  $BC(T)$  is a sound and complete optimization procedure for the combination of ILP with stably-infinite theories and that it generalizes the SMT framework. Our first IMT-inspired tool, CoBaSA, has been used to solve hard synthesis problems arising in the design of Boeing's 787. Our experience indicates that the IMT framework coupled with high-level modeling languages like CoBaSA has great potential for solving hard, industrial synthesis problems.

Many interesting research directions now open up. Among the more theoretical directions, it would be interesting to explore how we can relax the stably-infinite requirement on the background theory and still guarantee soundness and completeness. Our expectation is that there are fruitful connections between IMT and other formalisms and techniques, *e.g.*, SMT, constraint programming, domain-specific cut generation, and decomposition. For example, we believe that many instances of these techniques can be described more simply and elegantly using IMT.

The more practical research directions include implementing and experimenting with a general IMT solver based on  $BC(T)$ . This work is underway. We are planning to provide an SMT-LIB frontend, so that IMT can be used in place of an SMT solver; we outlined a strategy for mimicking SMT in Section 4. Future versions of CoBaSA will use our  $BC(T)$ -based solver as the backend. This will allow us to provide powerful, theory-based CoBaSA DSLs, *e.g.*, a comprehensive DSL for relational algebra, supported by a solver that uses efficient database techniques. A possible application is IMT-based SQL optimization.

## Acknowledgements

We would like to thank our colleagues Yue Huang and Mirek Riedewald, who provided the motivation for our database examples.

## References

- [1] Camlp5. See <http://pauillac.inria.fr/~ddr/camlp5/>.
- [2] CPLEX. See <http://www-01.ibm.com/software/integration/optimization/cplex-optimizer/>.
- [3] GAMS. See <http://www.gams.com/>.
- [4] T. Achterberg. *Constraint Integer Programming*. PhD thesis, Technische Universität Berlin, 2007.
- [5] C. Barrett, D. Dill, and A. Stump. Checking Satisfiability of First-Order Formulas by Incremental Translation to SAT. In *CAV*, 2002.
- [6] J. F. Benders. Partitioning procedures for solving mixed-variables programming problems. *Numerische Mathematik*, 4:238–252, 1961.
- [7] R. H. Byrd, A. J. Goldman, and M. Heller. Recognizing Unbounded Integer Programs. *Operations Research*, 35(1):140–142, 1987.
- [8] A. Cimatti, A. Franzen, A. Griggio, R. Sebastiani, and C. Stenico. Satisfiability Modulo the Theory of Costs: Foundations and Applications. In *TACAS*, 2010.
- [9] W. Cook, T. Koch, D. Steffy, and K. Wolter. An Exact Rational Mixed-Integer Programming Solver. In *IPCO*, 2011.
- [10] L. de Moura and N. Björner. Z3: An Efficient SMT Solver. In *TACAS*, 2008.
- [11] L. de Moura and H. Ruess. Lemmas on Demand for Satisfiability Solvers. In *SAT*, 2002.
- [12] B. Dutertre and L. de Moura. A Fast Linear-Arithmetic Solver for DPLL(T). In *CAV*, 2006.
- [13] B. Dutertre and L. de Moura. Integrating Simplex with DPLL(T). Technical report, SRI, 2006.
- [14] K. Dybvig, R. Hieb, and C. Bruggeman. Syntactic Abstraction in Scheme. *LISP and Symbolic Computation*, 5:295–326, 1993.
- [15] G. Faure, R. Nieuwenhuis, A. Oliveras, and E. Rodríguez-Carbonell. SAT Modulo the Theory of Linear Arithmetic: Exact, Inexact and Commercial Solvers. In *SAT*, 2008.
- [16] V. Ganesh and D. Dill. A Decision Procedure for Bit-vectors and Arrays. In *CAV*, 2007.
- [17] R. E. Gomory. Outline of an algorithm for integer solutions to linear programs. *Bulletin of the AMS*, 64:275–278, 1958.
- [18] M. Grötschel and O. Holland. Solution of large-scale symmetric travelling salesman problems. *Mathematical Programming*, 51:141–202, 1991.
- [19] C. Hang, P. Manolios, and V. Papavasileiou. Synthesizing Cyber-Physical Architectural Models with Real-Time Constraints. In *CAV*, 2011.
- [20] J. N. Hooker and G. Ottosson. Logic-Based Benders Decomposition. *Mathematical Programming*, 96, 2003.
- [21] D. Jovanovic and L. de Moura. Cutting to the Chase: Solving Linear Integer Arithmetic. In *CADE*, 2011.
- [22] T. Koch. *Rapid Mathematical Programming*. PhD thesis, Technische Universität Berlin, 2004.
- [23] E. Kohlbecker. *Syntactic Extensions in the Programming Language Lisp*. PhD thesis, Indiana University, 1986.
- [24] A. Kuehlmann, K. McMillan, and S. Sagiv. Generalizing DPLL to Richer Logics. In *CAV*, 2009.
- [25] Z. Manna and R. Waldinger. Fundamentals Of Deductive Program Synthesis. *IEEE TSE*, 18:674–704, 1992.
- [26] Z. Manna and C. Zarba. Combining Decision Procedures. In *10th Anniversary Colloquium of UNU/IIST*, 2002.
- [27] P. Manolios and V. Papavasileiou. Pseudo-Boolean Solving by Incremental Translation to SAT. In *FMCAD*, 2011.
- [28] P. Manolios, S. K. Srinivasan, and D. Vroon. BAT: The Bit-Level Analysis Tool. In *CAV*, 2007.
- [29] P. Manolios, D. Vroon, and G. Subramanian. Automating Component-Based System Assembly. In *ISSA*, 2007.
- [30] J. McCarthy. Towards a Mathematical Science of Computation. In *Congress IFIP-62*, 1962.
- [31] J. E. Mitchell. Branch-and-Cut Algorithms for Combinatorial Optimization Problems. In *Handbook of Applied Optimization*, pages 223–233. Oxford University Press, 2000.
- [32] G. Nelson and D. C. Oppen. Simplification by Cooperating Decision Procedures. *TOPLAS*, 1:245–257, 1979.
- [33] A. Neumaier and O. Shcherbina. Safe bounds in linear and mixed-integer linear programming. *Mathematical Programming*, 99:283–296, 2004.
- [34] R. Nieuwenhuis and A. Oliveras. Congruence Closure with Integer Offsets. In *LPAR*, 2003.
- [35] R. Nieuwenhuis and A. Oliveras. On SAT Modulo Theories and Optimization Problems. In *SAT*, 2006.
- [36] R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Solving SAT and SAT Modulo Theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T). *JACM*, 53(6):937–977, 2006.
- [37] Robert Fourer and David Gay and Brian Kernighan. *AMPL: A Modeling Language for Mathematical Programming*. Brooks/Cole Publishing Company, 2003.
- [38] Scott Cotton. Natural domain SMT: a preliminary assessment. In *FORMATS*, 2010.
- [39] R. Sebastiani and S. Tomasi. Optimization in SMT with LA(Q) Cost Functions. In *IJCAR*, 2012.
- [40] S. A. Seshia and R. E. Bryant. Deciding Quantifier-Free Presburger Formulas Using Parameterized Solution Bounds. In *LICS*, 2004.
- [41] C. Tinelli and M. Harandi. A New Correctness Proof of the Nelson–Oppen Combination Procedure. In *FroCoS*, 1996.
- [42] S. Tobin-Hochstadt, V. St-Amour, R. Culpepper, M. Flatt, and M. Felleisen. Languages as Libraries. In *PLDI*, 2011.